SANDIA REPORT

SAND2009-5515
Unlimited Release
Printed September 2009

Investigating Methods of Supporting Dynamically Linked Executables on High Performance Computing Platforms

James H. Laros III

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories

SAND2009-5515
Unlimited Release
Printed September 2009

Investigating Methods of Supporting Dynamically Linked Executables on High Performance Computing Platforms

James H. Laros III
1422 Scalable Computer Architectures
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
jhlaros@sandia.gov

Suzanne M. Kelly
1423 Scalable Systems Software
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
smkelly@sandia.gov

Michael J. Levenhagen
1423 Scalable Systems Software
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
mjleven@sandia.gov

Kevin Pedretti
1423 Scalable Systems Software
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
ktpedre@sandia.gov

Abstract

Shared libraries have become ubiquitous and are used to achieve great resource efficiencies on many platforms. The same properties that enable efficiencies on time-shared computers and convenience on small clusters prove to be great obstacles to scalability on large clusters and High Performance Computing platforms. In addition, Light Weight operating systems such as Catamount[9] have historically not supported the use of shared libraries specifically because they hinder scalability. In this report we will outline the methods of supporting shared libraries on High Performance Computing platforms using Light Weight kernels that we investigated. The considerations necessary to evaluate utility in this area are many and sometimes conflicting. While our initial path forward has been determined based on this evaluation we consider this effort ongoing and remain prepared to re-evaluate any technology that might provide a scalable solution.

Acknowledgment

The authors of this report would like to recognize the significant contributions of the author of both Statifier and Ermine, Valery Reznic. Reznic spent significant amounts of time not only responding to questions concerning his tools but on general issues surrounding dynamically linked executables and subtleties of Glibc and ELF. His assistance was invaluable.

Contents

Summary

This report is an evaluation of a range of possible methods of supporting dynamically linked executables on capability class[1] High Performance Computing platforms. Efforts are ongoing and extensive testing at scale is necessary to evaluate performance. While performance is a critical driving factor, supporting whatever method is used in a production environment is an equally important and challenging task.

[1]Systems designed to support applications that use a significant fraction of the total resource in support of a single cooperating application.

Nomenclature

Archive - See Static Library

Dynamically Linked Executable - Executables that contain references to external functions or variables that are contained in shared libraries. The shared objects referenced by the dynamically linked executable are mapped (linked into) into the executables memory space at run-time.

Dynamically Loaded Shared Object - Objects contained in shared libraries that are conditionally loaded, at runtime, into a dynamically linked executables memory space using the dlopen() API.

Library - A file built for the purpose of sharing code between applications. Libraries may be static or shared.

Loader - See Runtime Linker

Runtime Linker - An interpreter that accomplishes loading of the dynamically linked executable and resolves external references to objects contained in shared libraries and maps them into the executables memory space. Also Loader.

Shared Library - Libraries that contain objects that are linked into dynamically linked executables at runtime. These objects are referenced by dynamically linked executables at build (link) time.

Static Library - Libraries that contain objects that are linked into statically linked executables at initial build or link time. Also called Archives.

Statically Linked Executable - A standalone or self-sufficient executable. An executable that resolves externally referenced functions and or variables from static libraries and includes them into the final executable at build time (link time).

1 Introduction

In this report we discuss the results of evaluating the efficacy of a number of different methods of supporting dynamically linked executables on High Performance Computing (HPC) platforms, specifically those using a Light Weight Kernel (LWK) operating system. Catamount[9], and predecessors, have historically only supported launching statically linked executables. This choice was made largely for performance reasons. A statically linked executable can be launched on 12,960 nodes (38,400 cores) in 30 seconds. Efficiencies such as these are critical to the success of capability runtime systems.

While many of the terms we use in describing the methods evaluated are commonplace, they have unfortunately been sufficiently obfuscated and require clarification on how they will be used in this document. The term ***library*** is used to describe a method of sharing code (in the form of objects) between disparate applications. The term library does not indicate how the objects are used. (Libraries can also be used to segregate portions of a single application but this usage is peripheral to our discussion.) Libraries can be built to be used as ***static*** (also called archives) or ***shared***. Objects in static libraries are included into the final executable at link time, producing a stand-alone or self-sufficient final executable (***static or statically linked executable***). Only the objects needed by the final executable (also called application in this document) are linked in (included). The objects in shared libraries are referenced (not included) by an application at link time (***dynamically linked***). This reference, or note, indicates the application has a dependency on an object located in a specific shared library. It is important to note this reference will be resolved at run-time (the shared library containing the referenced object will be located and verified to be the correct version) by the ***runtime linker or loader*** and mapped into the application's memory space, but the shared object will not be loaded (into the application's memory space) until needed. In contrast to statically linked executables, dynamically linked executables are not self-sufficient since they depend on the presence of shared libraries. How this dependency is accommodated will be one of the topics of our discussion.

There is an additional method of using objects contained in shared libraries we discuss in this report. A ***dynamically loaded shared object*** is an object contained in a shared library that is externally referenced by a dynamically linked executable but is not resolved immediately at run-time. These shared objects may or may not be used by the application depending on application logic. Shared libraries requested in this way are accessed using what we will refer to as the ***dlopen() API*** . This method of accessing shared objects complicates an already challenging issue since each node or core participating in an application may arbitrarily (from the perspective of the runtime system) request an object from a shared library.

In the following sections we will discuss the methods evaluated, focusing primarily on their applicability to Catamount and future LWK's.

- Levenhagen Method (Section 2)
- Filesystem Method (Section 3)
- Statifier Method (Section 4)

- Ermine Method (Section 5)

The following evaluation criteria were used where applicable.

- Scalability
- Portability
- Standards based
- Production issues and Maintainability

How these criteria apply to each method will be discussed, where appropriate, and explained within the context of that method. Standalone conclusions will be included in each section describing a specific method. In Section 6 we will describe how a number of the challenges specific to Catamount have been mitigated by decisions made with our current LWK initiative. While many obstacles have been removed many scalability challenges remain. Comparative conclusions will be discussed in Section 7.

Definitions for specific terms are included in the Nomenclature section found in this report.

2 Levenhagen Method

Prior to this evaluation effort Michael Levenhagen developed a prototype, based on Catamount (specifically Catamount Virtual Node (CVN)), which was intended to demonstrate a method of supporting dynamically linked executables on an HPC platform (Red Storm[5]). The resulting prototype, coined the Levenhagen method in this report, modified the Catamount yod[2]/pct[3][4] load protocol in the following ways. The default behaviour of the load protocol upon executing yod is to memory map the executable (statically linked executable in the default case) and fan-out the executable to all participating nodes. (This protocol has proven to be very efficient and scalable.) To support dynamically linked executables it is necessary to have or duplicate the functionality of a runtime linker (loader). While a port of Glibc[3] is part of the standard Catamount tool-chain, the Catamount specific Glibc is dated and is only built to support statically linked executables. This presents a problem since the Glibc loader is only built when Glibc is built to support dynamically linked executables.

This effort included an evaluation of the level of effort involved in building the Catamount specific Glibc to support dynamically linked executables. This evaluation was accomplished by simply attempting this task. While significant progress was made in this direction, it was determined the level of effort required to produce a production ready version would, at a minimum, be twice the original porting effort, which was approximately three man months.

Since building the Catamount Glibc loader was determined to be impractical, at this time, and a loader, or the functionality of a loader, is necessary, other options had to be considered. An acceptable alternative, from the embedded world, was found in uClibc[8]. This open source Glibc compatible library provides a very large percentage of Glibc features in an extremely small number of lines of code. Producing a loader, sufficient for prototyping and testing based on uClibc compatible with Catamount, proved to be a relatively simple task.

Using the uClibc loader the default load protocol was modified to perform the following tasks: When yod is executed it analyzes the executable to determine if it is statically or dynamically linked. If statically linked, it reverts to the default protocol. If the executable is dynamically linked the load protocol maps both the uClibc loader and the dynamically linked application into memory on the launch node. Both the loader and the application are distributed by fanout to each participating node. In the default case, only the application is distributed to participating nodes.

At this point, both the loader and the executable (application) are memory resident on each node participating in the application. Each node begins execution but instead of simply starting the application, the loader executes the application with the goal of determining which shared libraries are necessary. (Recall, externally referenced functions are resolved at link time, the shared libraries

[2]Yod is the command line parallel application launcher, part of the Catamount runtime system (similar to the mpirun utility)

[3]Pct (Process Control Thread) is the managment thread that runs under the Catamount QK Kernel which manages compute resources for applications running on Catamount nodes

[4]Yod/pct protocol refers to the communication protocol used between yod and pct to, in this case, distribute application binaries to nodes participating in a job

that fulfill these requests are referenced in the ELF[5] executable (application). The loader interprets these references and requests resolution of each shared library.) Each node independently and sequentially requests each required shared library. An assumption is made that each executable will require the same shared libraries in the same sequence. This is a valid assumption for load time shared libraries (as opposed to shared libraries requested using the dlopen()[3] API). Each shared library is efficiently fanned out using the standard yod/pct protocol. Once all necessary shared libraries are distributed to all participating nodes, the loader does its work of symbol resolution and relocation on each node. When this is complete the application enters main(), on each node, and execution continues as normal.

In evaluating this method, we found this approach provides the scalability necessary for both existing and near term platforms at a minimum. However, issues not unique to this method that could potentially impact scalability remain. When an executable (application) requires a shared library, efficiencies are obtained on time-shared multi-user systems by sharing shared libraries. That is, in normal circumstances many applications executing in a single memory space will often use the same shared libraries. Efficiency is gained by only one copy of each shared library occupying memory space. In addition, only when an application requires an object from the shared library is that object loaded into memory (more specifically portions of a shared library are mapped in units called segments and only loaded when the page of memory mapped to that location is accessed). This allows for dynamically linked executables to remain small relative to a statically linked executables and resource efficiencies resulting from sharing common libraries.

Unfortunately, the execution environment is quite different on capability class HPC platforms (traditionally a space shared environment). Typically, a single application executes on each node. This precludes any efficiencies gained by resource sharing between applications. In addition, nodes typically have no locally accessible storage. If demand paging were to be used, accesses would be remote and likely very expensive. While the dynamically linked executable is likely to be smaller than the equivalent statically linked executable, the absence of demand paging capability requires each externally referenced shared library be loaded, in its entirety, into each nodes memory space. The result is the total memory burden for dynamically linked applications will far exceed the equivalent statically linked application. The Levenhagen method efficiently mitigates the challenge of distributing shared libraries to participating nodes. An important achievement. It does not, however, address some of the other inherent implications of supporting shared libraries on capability class platforms (i.e. memory inefficiencies).

The use of uClibc was a sane choice for the initial prototyping and testing effort but presents portability and standards concerns. If this method of supporting dynamically linked executables, specifically for Catamount, is chosen for production use, it should be recognized that uClibc does not assume to support all Glibc functionality. It should also be assumed that with further testing additional challenges related to Glibc support might be encountered. As with the Catamount specific Glibc port, any uClibc port would be very Catamount specific, therefore not widely portable. Additionally, choosing to use uClibc to provide the loader only addresses one of the requirements of complete support for dynamically linked executables. Catamount system libraries currently support only statically linked executables. One of the driving factors to support dynamically linked

[5]ELF (Executable and Linkable Format) common standard format for object code.

executables is the user community's desire to not have to re-link and or re-compile when changes are made to the Catamount system libraries. Providing shared library versions of these system libraries could be challenging. The level of effort to build shared Catamount system libraries was not estimated as part of this task.

The Levenhagen method, as designed, can provide limited support for the dlopen() API. If the dynamically linked application uses the dlopen() API in a consistent manner, that is each application follows the same logic path and requires the same shared library in the same sequence, the protocol described previously will satisfy dynamically loaded shared library requests. A more challenging aspect of supporting the dlopen() API is the very real possibility that not all participating nodes in the application will ask for the same shared library in the same sequence at the same time or at all. The Levenhagen method does not address this possibility. Supporting unpredictable dlopen() requests would be a challenge and require significant additional research and development.

Finally, while certainly not specific to this method in particular, supporting any custom solution in a production environment increases effort and requires significant maintenance commitments. Efficiencies could be gained by leveraging standards but due to intentionally imposed limitations of the Catamount runtime system, support of standard tool-chain components is impractical at this time.

3 File-system Method

This method of accessing shared libraries by dynamically linked executables, at its basis, is the standard method currently used on most platforms. For example, many cluster systems support dynamically linked executables by providing access to shared libraries on local disks (local to each node), remotely mounted filesystems (like NFS), and even by pre-populating memory filesystems with shared libraries. Unfortunately, each approach has inherent limitations. It is well established for capability class systems that the negative implications of local disk far outweigh any benefit and therefore render this option impractical. Likewise, providing access to remotely mounted filesystems for the purpose of accessing shared libraries is inefficient. As mentioned in Section 2, at runtime each reference to a shared library is resolved and memory mapped into each individual application's memory space. To support this first stage a filesystem would have to support tens of thousands of simultaneous *getattr()*[6] requests for a single file. Even if this could be done in a scalable manner when the application needs to actually use the object from the shared library the same tens of thousands of nodes would all come to a halt waiting for the shared library segment to be loaded into each of the tens of thousands of individual memory spaces from a single file location. Without mitigating these issues in some way this approach clearly will not scale to the node counts we anticipate. Taking the approach of simply loading all shared libraries an application *might* need into a memory based filesystem, at for instance boot time, would simply waste critical memory resources. Since each application runs on its own dedicated node the same inefficiencies described in Section 2 apply. Additionally, even if any of these approaches were practical, the challenges of providing a loader, a version of Glibc and Catamount shared system libraries would require the same effort as discussed in Section 2. If we ignore Catamount specific issues and focus on efficient delivery and access to shared libraries from each node, there are numerous methods that might be investigated.

Catamount uses a filesystem layer called libsysio[9],[7]. One approach would be to enable libsysio to support loading shared libraries to each node using the same yod/pct protocol described in Section 2. Libsysio could be modified to recognize file IO requests made for a specific path and to use the yod/pct protocol to efficiently distribute the shared library in its entirety to each participating node. This approach shares in all of the pros and cons previously described in Section 2 related to the yod/pct protocol.

The libsysio layer could be modified to use other methods that might provide efficient shared library distribution. Blue Gene/L[1], for example, uses a proxy based filesystem configuration to support dynamically linked executables. This method could be reproduced using the libsysio layer. In our estimate, this method could provide equivalent efficiencies to the Blue Gene/L method but not be as efficient as either the Levenhagen Method (Section 2) or libsysio leveraging a yod/pct type distribution protocol.

Providing support for dynamically linked executables, if we ignore the added complications of supporting the dlopen() API, could be achieved by pre-determining which shared libraries an application requires and then efficiently distributing those shared libraries to a memory based filesystem

[6]Function call to retrieve file attribute information.

on each node. This method closely resembles the method chosen by Cray Inc.^{TM}to support dynamically linked executables. The mechanisms to pre-determine which shared libraries are required is available with either standard tools like ldd[7] or simple to achieve by examining the executable (assuming it is an ELF executable). If we chose to pursue this method in a Catamount environment in addition to all of the previously discussed challenges, support for a memory based filesystem would have to be added along with support for mmap()[8] read()[9] and open()[10] which are currently not supported for this activity. From the perspective of supporting dunamically linked applications in the Catamount environment, filesystem approaches such as libsysio provide little if any advantage compared to the Levenhagen method and share most, if not all, of the challenges.

[7]Standard linker, part of Glibc

[8]Function call to map a file resident on storage other than memory into main memory

[9]Function call to read information from region addressed by file descriptor

[10]Function call which establishes a file descriptor to a specified file.

4 Statifier Method

Statifier[6] is an open source tool that packages a dynamically linked executable into a self-sufficient pseudo-static (statified) executable. The resulting statified executable contains the loader, the original dynamically linked executable and all shared libraries required by that executable. The initial reason this tool was attractive to investigate is the standard loader from Glibc could potentially be used to build the statified executable. In addition, symbol resolution and relocation is done during the statification process which has the potential to eliminate possible requirements for mmap(), read(), and open(). The final statified executable is a single binary executable. This could simplify the required changes to the yod/pct protocol to support dynamically linked executables. While the Levenhagen method proved to be a very scalable way to distribute the required shared libraries to participating nodes, it is logical to assume that distributing a single executable (default Catamount method) would be equally or more efficient. Statifier could also potentially support the dynamically linked executables that use the dlopen() API by specifying potentially used shared libraries with the LD_PRELOAD[11] environment variable.

To determine the feasibility of using Statifier to support dynamically linked executables we contrived a test case using a simple hello world type program provided in the Catamount distribution. The program was modified to reference a simple shared library of our construction. We built the final executable by statically linking with the Catamount system and Glibc libraries and dynamically linking with our simple shared library. We then statified the resulting dynamically linked executable to produce a statified executable. Our first problem was quickly evident. Catamount assumes that executables have only two PT_LOAD[12] segments. The statified executable in all cases contain greater than two PT_LOAD segments. Supporting more than two PT_LOAD segments would require similar but not quite as extensive modifications to the yod/pct load protocol as were described in Section 2. Properly placing all segments of the statified executable might prove more challenging and uncover additional complications.

Our efforts investigating the feasibility of Statifier were shortened due to very helpful and candid discussions with the tool's author. Simply stated, the author felt that Statifier might present support problems and was not appropriate for use in our production environment. The author considered Statifier a learning experience and had since used the experience gained in developing Statifier to start a new project with similar goals. This new project, Ermine[2], was also considered as a part of this project and will be discussed in Section 5. Since one of the most critical factors in evaluating each method was the ability to be supported in a production environment, using Statifier seemed a poor choice given the comments of the author.

Conceptually, Statifier satisfied many of our requirements. The scalability of distributing the single statified executable would likely be sufficient for our needs. While not previously mentioned, the statified object is very large. This should not be surprising since the final object contains the loader, the original dynamically linked executable and all referenced shared libraries. If the LD_PRELOAD option is used to include libraries potentially referenced by dlopen() calls the

[11]Environment variable used to specify search paths to search for referenced libraries
[12]In ELF, specifies a loadable segment

final executable would be even larger. Regardless, distributing a single large executable would likely be as, or more, efficient as distributing a number of objects, which in total, equal the size of the single statified object. Statifier uses the standard Glibc loader and shared libraries, therefore would likely be a highly portable solution. Additionally, while Statifier itself is not a standard it is open-source and uses standard tools to accomplish its goals.

5 Ermine Method

Ermine[2] was developed by the same author as Statifier. Ermine is a completely separate project sharing no code with Statifier. Ermine, in contrast to Statifier, is intended to be production quality and supports additional features Statifier does not. For the purposes of this discussion Ermine provides basically the same benefit, a single self-sufficient executable constructed from the loader, a dynamically linked executable and all referenced shared libraries. While the target audience for both Statifier and Ermine is not the HPC environment, these features are attractive and share all of the benefits outlined in Section 4. One additional feature of Ermine is worth mentioning; Ermine has the ability to pack arbitrary data files into the final executable package. This feature could be useful in the HPC environment to distribute input data files, for example. In Section 4 we discussed the necessity of modifying the yod/pct protocol to support greater than two PT_LOAD segments. Ermine produces only two PT_LOAD segments and might be more easily supported by the existing yod/pct protocol.

Ermine, however, does require support for mmap(), open() and read() which would require modification to the Catamount environment. Further analysis of the practicality of Ermine became academic due to the fact that Ermine is a closed source commercial product. Our criteria of standards based implies open source availability. While a solution like Ermine might be considered to support dynamically linked executables on Catamount if the effort were minimal (which it would not be) requiring a closed source commercial product as part of an open source LWK distribution, for example Kitten[4], is unacceptable.

6 Future Light Weight Kernels

In 2005 work on a next generation LWK began. This new effort, Kitten [13][4] is an open-source operating system designed specifically for HPC. Kitten borrows heavily from the Linux^{TM} code base, but in areas critical to scalability and performance like memory management and task scheduling, is written from scratch incorporating lessons learned from a now long history of LWK development at Sandia.

In Section 2 a number of dependencies specific to Catamount were mentioned that complicated support of dynamically linked executables. By design, Kitten avoids many of these complications. Kitten provides partial Linux API and ABI compatibility so that standard compiler tool-chains and system libraries (e.g., Glibc) can be used without modification. The resulting ELF executables can be run on either Linux or Kitten unchanged. It can be generally stated that the scalability challenges of supporting dynamically linked executables with Kitten are the result of features inherent to shared library support rather than complications resulting from trade-offs made in support of scalability and performance.

The lessons learned from both the initial prototyping and later analysis of the Levenhagen method, along with our other analysis, have provided an initial path forward for scalable dynamically linked executable support on Kitten. In short, since the design of Kitten mitigates virtually all of the challenges identified in Section 2, the scalability of the distribution protocol provides an attractive template to be used in the design of a Kitten launch protocol. It is important to note that additional unresolved challenges remain in supporting applications that wish to leverage the dlopen() API. Kitten, like Catamount, is a very flexible framework that allows rapid prototyping. A decision to support run-time shared library distribution through the launch protocol does not prohibit using a completely separate mechanism to support distribution of shared libraries requested using the dlopen() API.

[13]The name Kitten continues the cat naming theme, but indicates a new beginning.

7 Conclusion

In Section 2 we outlined a method designed to support dynamically linked executables that provides a scalable distribution of the shared libraries required by the application. While many challenges specific to Catamount led us to conclude that the modifications required, production and maintenance implications were prohibitive, the basic design proved worthy and is targeted for use in ongoing LWK development (Section 6). Section 4 and 5 provided interesting approaches that had great potential in solving issues specific to Catamount and for future LWK development. Unfortunately, both of these specific tools failed to meet one or more of our critical requirements. While our analysis determined these specific tools could not be used, the basic concept of distributing a self-sufficient executable package is attractive. Run-time distribution of a single executable has proved to be a very scalable approach and other than the size, both the Statifier and Ermine method provide the equivalent of a single executable for distribution. Techniques like this may warrant additional consideration. The Filesystem Method, described in Section 3, also remains a topic for future consideration. This method, in most instances, can be pursued independent of operating system considerations. As a result, one of the attractive aspects of this method is the inherent portability.

Table 1 provides a summary of the issues considered for each presented solution. The qualitative ratings of poor, fair, good and very good are in relation to each other, rather than as a general statement of textitgoodness.

Table 1. Matrix of key issues for support of dynamically linked executables versus potential solution methods in capability class HPC systems.

Issues/Method	Levenhagen	In-Memory Filesystem	Statifier	Ermine
Scalability	Very Good	Good	Very Good	Very Good
Portability	Poor	Fair	Good	Very Good
Standards Based	Poor	Fair	Good	Very Good
Maintainability	Requires Local Support	Requires Enhanced libsysio	Declared Insufficient	Source Code Not Available
Supports dlopen()	Poor	Yes	Yes	Yes
Larger Memory Requirements (Relative to static)	Yes	Yes	Yes	Yes
Custom glibc (Catamount)	Yes	Yes	Maybe	No
Custom mmap() (Catamount)	Yes	Yes	No	Yes

While our analysis determined that supporting dynamically linked executables, specifically in the Catamount production environment, required an unacceptable level of effort and subsequent maintenance, the potential convenience to the user community of this feature has made it a priority for future LWK implementations.

References

[1] Blue-Gene/L. Electronic reference, Lawrence Livermore National Labs, https://asc.llnl.gov/computing_resources/bluegenel.

[2] Ermine. Electronic reference, Open Source, http://www.magicermine.com.

[3] Gnu c library. Electronic reference, Open Source, http://www.gnu.org/software/libc/.

[4] Kitten. Electronic reference, Open Source, http://software.sandia.gov/trac/kitten.

[5] Redstorm. Electronic reference, Sandia National Laboratories, http://www.cs.sandia.gov/platforms/RedStorm.html.

[6] Statifier. Electronic reference, Open Source, http://statifier.sourceforge.net.

[7] sysio. Electronic reference, Open Source, http://www.sourceforge.net/libsysio.

[8] uclibc. Electronic reference, Open Source, http://www.uclibc.org.

[9] Suzanne M. Kelly and Ron Brightwell. Software Architecture of the Light Weight Kernel, Catamount . In *proceedings of the Cray User Group (CUG)*, 2005.

DISTRIBUTION:

1	MS	1319	James H. Laros III, 01422
1	MS	1319	Suzanne M. Kelly, 01423
1	MS	1319	John VanDyke, 01423
1	MS	1319	Kevin Pedretti, 1423
1	MS	1319	Micheal Levenhagen, 1423
1	MS	1319	Ronald Brightwell, 1423
1	MS	1319	James Ang, 1422
1	MS	0899	Technical Library, 9536 (electronic)

Sandia National Laboratories